IN-61
20328

NASA Contractor Report 187568

ICASE INTERIM REPORT 17

A MANUAL FOR PARTI RUNTIME PRIMITIVES
Revision 1

Raja Das
Joel Saltz
Harry Berryman

NASA Contract No. NAS1-18605
May 1991

(NASA-CR-187568) A MANUAL FOR PARTI RUNTIME PRIMITIVES, REVISION 1 Final Report (ICASE) 52 p
CSCL 09B
N91-25644
Unclas
G3/61 0020328

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

ICASE INTERIM REPORTS

ICASE has introduced a new report series to be called ICASE Interim Reports. The series will complement the more familiar blue ICASE reports that have been distributed for many years. The blue reports are intended as preprints of research that has been submitted for publication in either refereed journals or conference proceedings. In general, the green Interim Report will not be submitted for publication, at least not in its printed form. It will be used for research that has reached a certain level of maturity but needs additional refinement, for technical reviews or position statements, for bibliographies, and for computer software. The Interim Reports will receive the same distribution as the ICASE Reports. They will be available upon request in the future, and they may be referenced in other publications.

Robert G. Voigt
Director

A Manual for PARTI Runtime Primitives Revision 1

Raja Das and Joel Saltz and Harry Berryman*

Institute for Computer Applications in Science and Engineering,
NASA Langley Research Center,
Hampton VA 23065

Computer Science Department,
Yale University,
New Haven CT 06520

Abstract

Primitives are presented that are designed to help users efficiently program irregular problems (e.g. unstructured mesh sweeps, sparse matrix codes, adaptive mesh partial differential equations solvers) on distributed memory machines. These primitives are also designed for use in compilers for distributed memory multiprocessors. Communications patterns are captured at runtime, and the appropriate send and receive messages are automatically generated.

*ICASE, NASA Langley Research Center, Hampton, Virginia. Supported by NASA Contract No. NAS1-18605 while the author was in residence at ICASE, and by NSF grant ASC-8819374

PRECEDING PAGE BLANK NOT FILMED

1 Did Somebody Say PARTI?

1.1 Overview

PARTI stands for "Parallel Automated Runtime Toolkit at ICASE." Development of PARTI has been carried out at Yale University as well as ICASE and hence has been referred to as "PARTY" in some earlier papers. The PARTI runtime primitives are designed to help users to efficiently program loops found in irregular problems (e.g. unstructured mesh sweeps, sparse matrix codes, adaptive mesh partial differential equations solvers). These primitives are also designed for use in compilers for distributed memory multiprocessors. In the context of the PARTI project, we are also developing a variety of other tools including compilers for distributed machines. These primitives are some of the basic building blocks we are using in our efforts.

The primitives in this distribution run on any of the iPSC/2 or iPSC/860 machines produced by Intel Scientific Computing. They could easily be modified to run on most distributed memory machines. This document describes the operation of the PARTI primitives and gives several examples of how to use them. The rationale of the PARTI system (the PARTI line, as it were) was presented in [2] and summarized in [4]. The mechanisms incorporated in these primitives have been outlined in [2], [5], [4]. PARTI has been used in a variety of applications, including sparse matrix linear solvers, adaptive computational fluid dynamics codes, and in a prototype compiler [4] aimed at distributed memory multiprocessors.

1.2 Primitives Available in the Release

The PARTI system is divided into several levels. Level 0 primitives allow processors to access the distributed memory of a multiprocessor with a modicum of convenience. Level 1 primitives bind mapping information to arrays. This allows the user to store and manipulate constructs that describe multiprocessor mappings of distributed multidimensional arrays. Included with this distribution are the level 0 primitives outlined next.

The level 0 *scatter* allows each processor of a distributed memory machine to move data to off-processor memory locations. The level 0 *gather* allows each processor to obtain copies of data from memory locations in other processors. Level 0 primitives are provided to support initialization and access of distributed translation tables. Such distributed tables allow a user to assign globally numbered indices to processors in an irregular pattern. By using a distributed translation table, it is possible to avoid

replicating records of where distributed array elements are stored in all processors. Level 0 primitives also carry out off-processor accumulations; e.g. any processor can add to the contents of an off-processor memory location.

1.3 Primitives that exist but are not yet distributed

There are additional level 0 primitives not included with this release that support local caching of copies of off-processor data. These Level 0 primitives are presented in [3] and will be available in future PARTI releases. Level 1 primitives, also not available with this release, allow users to specify how distributed arrays are to be mapped onto sets of processors. The level 1 primitives support read, write and accumulate accesses to these mapped multidimensional arrays. The level 1 primitives also allow users to dynamically remap distributed arrays. The Level 1 primitives are described in [1]. It should be noted that use of PARTI primitives do not interfere with access to traditional message passing communications primitives. In particular, a user can call all of the iPSC supplied routines when using PARTI.

2 Installation

2.1 Getting PARTI

PARTI can be had in either several shar files or one tar file. The tar file is in general more convinient, but the shar files can be sent through the mail. PARTI can be obtained by anonymous ftp from *ra.cs.yale.edu*, from netlib, or by contacting:

```
Raja Das
ICASE
Mail Stop 132C
NASA Langley Research Center
Hampton, Va 06511
(804) 864-8004
raja@icase.edu
```

If you have the PARTI tar file, just change to the directory where you wish to put the PARTI subdirectory and type:

```
tar xof parti.tar
```

If you have the shar files, things are only mildly worse. You need the following files: docs.shar, free.shar, matmult.shar, papers.shar, src.shar, tests.shar, unst.shar and a makefile (called "makefile", oddly enough.) Put these files in the directory where you want the PARTI subdirectory and type

```
make unshar
```

2.2 Building PARTI

Either of the above installation procedures should create the following directory structures:

parti/docs documentation in latex, postscript and plain text

parti/examples/matmult sparse matrix multiplication described in Section B

parti/examples/unst sweep over unstructured mesh, described in section A.

parti/examples/free a conjugate gradient linear equation solver cg.c and cg_host.c not discussed in this documentation. (Free prize included in every copy of PARTI!). Also included is simple.c, a simple example involving several of the primitives.

parti/papers some of the relevant papers

parti/src source for the PARTI primitives

parti/tests test programs to verify correct installation

A makefile should be present in the PARTI directory. At the beginning of this makefile are several macros to be modified by the user.

NFLAG This macro is passed to the C compiler and linker when compiling and/or linking node programs. It should have one of the following values:

-node -sx for iPSC/2 machines with weitek floating point accelerators

-node -i860 for iPSC/860 machines

-node for vanilla iPSC/2 machines

NARC This macro indicates the archive to be used in creating the PARTI library. It should be set to one of the following:

ar for any iPSC/2

ar860 for an iPSC/860

LIB This macro should be set to the directory where the party library will be installed. It is prudent to use the full path name here. This directory must exist before the system is installed.

INCL This macro should be set to the directory where the PARTI include files will reside. It is prudent to use the full path name here. This directory must exist before the system is installed.

NPROCS This indicates the largest number of processors that the tests should be run on. Eight and sixteen are good values.

NODECC This macro should be set to the C compiler which will compile the node programs. The default compiler (cc) is always a correct choice. The pgcc compiler may also be used where appropriate.

NODEF77 This macro should be set the Fortran compiler to be used to compile the node programs. The default compiler (f77) is always a correct choice. The pgf77 compiler may be used where appropriate.

Make sure that the directories pointed to by `LIB` and `INCL` exist. If they do not, any attempt to install the party system there will fail. There are several objects to make. Typing the following make commands in the listed order should be sufficient to install and check the PARTI system on your computer.

make will compile the PARTI library but not install it in the designated directories.

make install will install the PARTI system in the designated directories.

make clean will remove object and executable file from various subdirectories.

make test will run several tests to see if everything has been compiled correctly.

3 Function Descriptions

3.1 Header Files

There are two header files which go with the PARTI library. The first is `parti.h`. This file contains the definitions of all structures, macro definition and function definitions needed to run the PARTI primitives. *It must be included in all C programs that use the PARTI system.* The second include file, `parti_more.h`, is used only when the system is compiled. It defines such things as message types, and static buffer lengths. It should not be necessary to include this file in applications which use PARTI. No header files need be included in Fortran applications.

Two of the primitives `schedule` and `build_translation_table` are functions that carry out preprocessing. `schedule` and `build_translation_table` allocate elements of structures `schedule_struct` and `trans_table` and then return pointers to structures. The above structures are defined in parti.h; macro definitions define `struct schedule_struct` as SCHED and define `struct trans_table` as TTABLE. parti.h also defines macros STRIPED and BLOCKED used in the procedure `build_translation_table`

3.2 Level 0 primitives

Level 0 gathers and scatters are accomplished by using three routines: *Scheduler*, *Gather*, and *Scatter*.

Scheduler on each processor is passed a list of indices K_j into `aloc` on each processor `j`. Scheduler produces a schedule `S` that controls the data that are to be fetched off-processor by *Gather* or scattered off-processor by *Scatter* .

On each processor, Gather inputs

1. a buffer into which the fetched elements are to be placed
2. a pointer to local array `aloc`
3. the schedule `S` produced by Scheduler

In Fig. 1 we introduce a running example to illustrate the *Scheduler*, *Gather* and *Scatter*. In this example we have three processors, each processor is passed a set of off-processor indices.

Gather executes sends and receives that fetch from processor `j` the appropriate elements from the array `aloc` on processor `j`. Then it places these elements into

Figure 1: Scheduler Example

Scheduler:

inputs list of indices on each processor

outputs a schedule `S`

E.g.

processor 1: (processor 2, index 5), (processor 3, index 7)

processor 2: (processor 1, indices 4, 5, 6), (processor 3 index 2)

processor 3: (processor 1, index 1),(processor 2 indices 1, 3, 4)

the user-supplied buffer. Fig. 2 continues the running example begun in Fig. 1. On processor `j` the array `aloc` is initialized as `aloc(i)` = `j * 100 + i` for `1` $\leq$ `i`. We depict the contents of `buffer` on each processor after Gather is executed.

Scatter is passed

1. a buffer from which each scattered datum is to be obtained
2. a pointer to local array `aloc`
3. the schedule `S` produced by Scheduler

Scatter executes sends and receives that put on processor `j` the appropriate elements from the buffer. Then Scatter places these elements into the appropriate elements of array `aloc` on processor `j`. Fig. 3 continues the running example. We assume that on processor `j`, we initialize `buffer` as `buffer(i)` = `j * 100 + i` for `1` $\leq$ `i`, we initialize `aloc` so that `aloc(i)` = `0`. After Scatter executes, we depict, on each processor `j` the contents of `aloc`.

3.2.1 Functioning of the Scheduler, Gather and Scatter

Both the procedures *Scatter* and *Gather* have three stages. They permute data into buffers to be sent. They perform the needed communication, then they perform another permutation.

Figure 2: Gather Example

Gather:

inputs schedule `S` produces by *Scheduler*

inputs pointer to local array `aloc` from which gathered elements are to be fetched

outputs fetched elements placed in local array `buffer`

E.g. assume

processor 1: $\texttt{aloc(i)} = \texttt{100} + \texttt{i}\ ,\ \texttt{1} \leq \texttt{i}$

processor 2: $\texttt{aloc(i)} = \texttt{200} + \texttt{i}\ ,\ \texttt{1} \leq \texttt{i}$

processor 3: $\texttt{aloc(i)} = \texttt{300} + \texttt{i}\ ,\ \texttt{1} \leq \texttt{i}$

Gather returns:

buffer	Processor 1	Processor 2	Processor 3
1	205	104	101
2	307	105	201
3	-	106	203
4	-	302	204

Figure 3: Scatter Example

Scatter:

inputs schedule `S` produces by *Scheduler*

inputs elements to be scattered, these are placed in local array `buffer`

outputs scattered elements, these are placed in local array `aloc`

E.g. assume

processor 1: `buffer(i)` $= 100 +$ `i` , $1 \leq$ `i`

processor 2: `buffer(i)` $= 200 +$ `i` , $1 \leq$ `i`

processor 3: `buffer(i)` $= 300 +$ `i` , $1 \leq$ `i`

processor 1: `aloc(i)` $= 0$, $1 \leq$ `i`

processor 2: `aloc(i)` $= 0$, $1 \leq$ `i`

processor 3: `aloc(i)` $= 0$, $1 \leq$ `i`

After Scatter is called:

aloc	Processor 1	Processor 2	Processor 3
1	301	302	0
2	0	0	204
3	0	303	0
4	201	304	0
5	202	101	0
6	203	0	0
7	0	0	102

The scheduler first determines how many messages each processor must send and receive during the data exchange phase. Defined on processor j is an array $\texttt{nmsgs}^{\texttt{j}}$. Processor j sets the value of $\texttt{nmsgs}^{\texttt{j}}(\texttt{i})$ to 1 if it needs data from processor i or to 0 if it does not. The scheduler then replaces $\texttt{nmsgs}^{\texttt{j}}$ with the element-by-element sum $\texttt{nmsgs}^{\texttt{j}}(\texttt{i}) \leftarrow \sum_k \texttt{nmsgs}^{\texttt{k}}(\texttt{i})$. This operation utilizes a function that imposes a fan-in tree to find the sums. Since the resulting sum is kept in $\texttt{nmsgs}^{\texttt{j}}$, at the end of the fan-in on every processor, $\texttt{nmsgs}^{\texttt{j}}(\texttt{i})$ is the number of messages that processor must send during the exchange phase. Next, each processor sends a *request list* to every other processor. The request list sent from processor p to processor q contains the indices of data needed by processor p that are stored on processor q.

The number of non-empty request lists each processor will receive is equal to the number of messages that the processor will send in the gather or scatter phase. Each request list is placed in an array indexed by the processor from which the list came. When the scheduler is finished, each processor has an array of request lists obtained from other processors. The $\texttt{j}^{\texttt{th}}$ element of this array contains the request list obtained from processor j. At this point in the execution, each processor i knows which elements of aloc local to processor i that must be sent to other processors. This information is used to generate the schedule S of pairs of send and receive statements. These send/receive pairs will exchange the requested data for either a gather or a scatter. The gather or the scatter is passed the schedule S with the required buffer space. It then carries out the required communication.

3.3 schedule()

This procedure carries out the preprocessing needed for carrying out optimized gather exchanger and scatter exchanger routines. Every processor must participate in this procedure call. On each processor, a schedule is passed a list of processors and local indices from which a gather procedure on that processor can later obtain data (or to which a scatter procedure on that processor can later write data). schedule returns a pointer to a structure of type SCHED, this pointer is used in gather, scatter and scatter_FUNC operations (Sections 3.4, 3.5, 3.6).

Synopsis

SCHED *schedule(local,proc,ndata)

Parameter declarations

int *local local index to be gathered from or scattered to

int *proc processors to be gathered from or scattered to

int ndata number of data involved in gather or scatter

Return value

Returns pointer to structure of type SCHED which can be used in PREFIXgather, PREFIXscatter, PREFIXscatter_add, PREFIXscatter_sub, PREFIXscatter_mult.

Example

Node 0 schedules a fetch of elements 1 and 2 from a (so far unspecified) array on node 1; node 1 schedules a fetch of element 1 from an array on node 0 and 0 from an array on node 1.

```
int local[2], proc[2], ndata;
SCHED *schedinfo;



if(mynode()==0){
  proc[0] = 1;
  local[0] = 1;
  proc[1] = 1;
  local[1] = 2;
  ndata = 2;
}

if(mynode()==1){
  proc[0] = 0;
  local[0] = 1;
  proc[1] = 1;
  local[1] = 0;
  ndata = 2;
}
```

```
schedinfo = schedule(local,proc,ndata);
```

3.4 PREFIXgather()

PREFIX can be d (double precision), i (integer) , f (floating point) or c (character) This procedure is the gather exchanger procedure described above and in [1]. PREFIXgather uses a schedule produced by a call to schedule, the schedule is passed to PREFIXgather in structure SCHED schedinfo. Copies of data values obtained from other processors are placed in memory pointed to by buffer. Also passed to PREFIX gather is a pointer to the location from which data is to be fetched *on the calling processor.* This pointer is designated here as aloc, aloc corresponds to $aloc^i$ above and in [1].

Synopsis

void PREFIXgather(schedinfo,buffer,aloc)

Parameter Declarations

SCHED *schedinfo information obtained from schedule's preprocessing of reference pattern

TYPE *buffer pointer to buffer for copies of gathered data values

TYPE *aloc location from which data is to be fetched from calling processor

Return Value

None

Example

We assume that schedule has already been called with the parameters presented in Section 3.3. Our example will assume that we wish to gather double precision numbers, i.e. that we will be calling dgather. On each processor, *aloc points to the arrays from which values are to be obtained. *buffer points to the location into which will be placed copies of data values obtained from other processors.

```
double buffer[2], aloc[3];
SCHED *schedinfo;


 for(i=0;i<3;i++){
   aloc[i] = mynode() + 0.1*i;
 }

 dgather(schedinfo,buffer,aloc);
```

On processor 0, buffer[0] and buffer[1] are now equal to 1.1 and 1.2. On processor 1, buffer[0] and buffer[1] are now equal to 0.1 and 1.0.

3.5 PREFIXscatter()

PREFIX can be d (double precision), i (integer) , f (floating point) or c (character). This procedure is the scatter exchanger procedure described above and in [1]. PREFIXscatter uses a schedule produced by a call to schedule, the schedule is passed to PREFIXscatter in structure SCHED schedinfo. Copies of data values to be scattered to other processors are placed in memory pointed to by buffer. Also passed to PREFIX scatter is a pointer to the location to which copies of data are to be written *on the calling processor.* This pointer is designated here as aloc, aloc corresponds to $aloc^i$ above and in [1].

Synopsis

void PREFIXscatter(schedinfo,buffer,aloc)

Parameter Declarations

SCHED schedinfo information obtained from schedule's preprocessing of reference pattern

TYPE *buffer points to data values to be scattered from a given processor

TYPE *aloc points to first memory location on calling processor for scattered data

Return Value

None

Example

We assume that schedule has already been called with the parameters presented in Section 3.3. Our example will assume that we wish to scatter double precision numbers, i.e. that we will be calling dscatter. On each processor, *aloc points to the arrays to which values are to scattered. *buffer points to the location from which will be obtained data that will be scattered The processor and local_array index to which the values are to be scattered was designated during an earlier call to schedule.

```
double buffer[2], aloc[3];
SCHED *schedinfo;


for(i=0;i<3;i++){
  aloc[i] = 10.0;
}

if(mynode()==0){
  buffer[0] = 444.44;
```

```
  buffer[1] = 555.55;
}

if(mynode()==1){
  buffer[0] = 666.66;
  buffer[1] = 777.77;
}

dscatter(schedinfo,buffer,aloc);
```

On processor 0, the first three elements of aloc are 10.0, 666.66 and 10.0. On processor 1, the first three elements of aloc are 777.77, 444.44 and 555.55.

3.6 PREFIXscatter_FUNC()

PREFIX can be d (double precision), i (integer) , f (floating point) or c (character). FUNC can be add, sub or mult . PREFIXscatter stores data values to specified locations. PREFIXscatter_FUNC allows one processor to specify computations that are to be performed on the contents of given memory location of another processor. The procedure is in other respects analogous to PREFIXscatter.

Synopsis

void PREFIXscatter_FUNC(schedinfo,buffer,aloc)

Parameter Declarations

SCHED *schedinfo information obtained from schedule's preprocessing of reference pattern.

TYPE *buffer points to data values that will form operands for the specified type of remote operation.

TYPE *aloc points to first memory location on calling processor to be used as targets of remote operations.

Return Value

None

Example

We assume that schedule has already been called with the parameters presented in Section 3.3. Our example will assume that we wish to scatter and add double precision numbers, i.e. that we will be calling dscatter_add. On each processor,

*aloc points to the arrays to which values are to be scattered and added. *buffer points to the location from which will be obtained the values to be scattered and added. The processor and local_array index to which the values are to be scattered and added was designated during an earlier call to schedule.

```
double buffer[2], aloc[3];
SCHED *schedinfo;


 for(i=0;i<3;i++){
   aloc[i] = 10.0;
 }

 if(mynode()==0){
   buffer[0] = 444.44;
   buffer[1] = 555.55;
 }

 if(mynode()==1){
   buffer[0] = 666.66;
   buffer[1] = 777.77;
 }

 dscatter_add(schedinfo,buffer,aloc);
```

On processor 0, the first three elements of aloc are 10.0, 676.66 and 10.0. On processor 1, the first three elements of aloc are 787.77, 454.44 and 565.55.

3.7 build_translation_table()

In order to allow a user to assign globally numbered indices to processors in an irregular pattern, it is useful to be able to define and access a distributed translation table. By using a distributed translation table, it is possible to avoid replicating records of where distributed array elements are stored in all processors. The distributed table is itself partitioned in a very regular manner. A processor that seeks to access an element I of a irregularly distributed data array is able to compute a simple function that designates a location in the distributed table; the location of the actual array element sought is obtained from the distributed table.

The procedure build_translation_table constructs a distributed translation table. It assumes that distributed array elements are globally numbered. Each processor passes build_translation_table a set of indices for which it will be responsible. The distributed translation table may be striped or blocked across the processors. With a striped translation table, the translation table entry for global index I is stored in processor (I modulo number_of_processors); the local index of the translation table is (I/ number_of_processors). In a blocked translation table, translation table entries are partitioned into a number of equal sized ranges of contiguous integers, these ranges are placed in consecutively numbered processors. With blocked partitioning, the block corresponding to index I is (I/B) and the local index is (I modulo B), where B is the size of the block. Let M be the maximum global index passed to build_translation_table by any processor and NP represent the number of processors; B = $\lceil M/NP \rceil$.

build_translation_table returns a pointer to a structure of type TTABLE; this pointer is used in dereference, defined in section 3.8.

Synopsis

TTABLE *build_translation_table(part,indexarray,ndata)

Parameter Declarations

int part how translation table will be mapped - may be BLOCKED or STRIPED

int *indexarray each processor P specifies list of globally numbered indices for which P will be responsible

int ndata number of indices for which processor P will be responsible

Return Value

structure of type TTABLE; this structure contains a given processor's portion of the distributed translation table

Example

An example to demonstrate the use of both build_translation_table and dereference can be found in Section 3.8.

3.8 dereference()

dereference accesses the distributed translation table constructed in build_translation_table. dereference is passed a pointer to a structure of type TTABLE; this structure defines the irregularly distributed mapping created in procedure build_translation_table. dereference is passed an array with global indices that need to be located in distributed memory; dereference returns arrays local and proc that contain the processors and local indices corresponding to the global indices.

Synopsis

void dereference(index_table,global,local,proc,ndata)

Parameter declarations

int *global list of global indices we wish to locate in distributed memory

int *local local indices obtained from the distributed translation table that correspond to the global indices passed to dereference

int *proc array of distributed translation table processor assignments for each global index passed to dereference

Table 1: Values obtained by dereference

Processor	proc[0]	local[0]	proc[1]	local[1]
0	0	0	1	0
1	1	1	0	1

int ndata number of elements to be dereferenced

TTABLE *index_table distributed translation table datastructure created in build_translation_table

Return value

None

Example

A one dimensional distributed array is partitioned in some irregular manner so we need a distributed translation table to keep track of where one can find the value of a given element of the distributed array.

In the example below, we initialize a translation table. Processor 0 calls build_translation_tabl and assigns indices 0 and 3 to processor 0, processor 1 calls build_translation_table and assigns indices 1 and 2 to processor 1. The translation table is partitioned between processors in blocks.

Processor 0 then uses the translation table to dereference global variables 0 and 1, processor 1 uses the translation table to dereference global variables 2 and 3. On each processor, dereference carries out a translation table lookup. The values of proc and local are returned by dereference are shown in Table 1). The user gets to specify the processor to which each global index is assigned, note however that build_translation_table assigns local indices.

```
#include <stdio.h>
#include "parti.h"

main()
{
  int size, i, *index_array;
```

```
  int *deref_array;
  int *local, *proc;
  TTABLE *table;


  size = 2;
  index_array = (int *) malloc(sizeof(int)*size);
  deref_array = (int *) malloc(sizeof(int)*size);
  local = (int *) malloc(sizeof(int)*size);
  proc = (int *) malloc(sizeof(int)*size);

/*Assign indices 0 and 3 to processor 0 */
if(mynode()==0)
    {
    index_array[0] = 0;
    index_array[1] = 3;
    }
/*Assign indices 1 and 2 to processor 1 */
if(mynode()==1)
    {
    index_array[0] = 1;
    index_array[1] = 2;
    }

  /* set up a translation table */

  table = build_translation_table(BLOCKED,index_array,size);


/* Processor 0 seeks processor and local indices
for global array indices 0 and 1 */
if(mynode()==0)
    {
    deref_array[0] = 0;
    deref_array[1] = 1;
    }
/* Processor 1 seeks processor and local indices
```

```
for global array indices 2 and 3 */
if(mynode()==1)
    {
    deref_array[0] = 2;
    deref_array[1] = 3;
    }

 /* Dereference a set of global variables */

  dereference(table,deref_array,local,proc,size);

/* local and proc return the processors and local indices where
global array indices are stored.
In processor 0, proc[0] = 0, proc[1] = 1, local[0] = 0 , local[1] = 0;
In processor 1, proc[0] = 1, proc[1] = 0, local[0] = 1 , local[1] = 1;
*/
}
```

Now assume that processor 0 needs to know to values of distributed array elements 0,1, and 3 while processor 1 needs to know the value of element 2. We call dereference to find the processors and the local indices that correspond to each global index. At this point schedule can be called and gathers and scatters carried out.

3.9 localize()

When loops access data residing off processor, some pre-processing is necessary before these loops can be executed. The pre-processing involves setting a schedule to bring in the off-processor data, and changing all the global references to local ones. The primitive `localize` makes calls to `dereference` and `schedule` to do all the necessary processing. The schedule pointer returned by `localize` is used to gather data and store it at the end of the local array. This schedule pointer is created such that multiple copies of the same data is not brought in during the gather phase. The elemination of duplicates is achieved by using a hash table. `Localize` returns the local reference string corresponding to the global references which are passed as a

parameter to it. The number of off processor data elements are also returned by `localize` so that one can allocate enough space at the end of the local array.

Synopsis

void localize(tabptr,lsched,global_refs, local_refs,ndata,n_off_proc,my_size)

Parameter Declarations

TTABLE *tabptr pointer to the distributed translation table, build for the local array being dealt with.

SCHED **lsched pointer to the data structure for schedule, which stores all the send receive information (returned by `localize`).

int *global_refs pointer to the array which stores all the global reference string.

int *local_refs pointer to the array which stores the local reference string corresponding to the global references (returned by `localize`).

int ndata number of global references.

int *n_off_proc address of the number of off processor data (returned by `localize`).

int my_size the size of my local array.

Return Value

None

Example

Nodes 0 and 1 takes part in a computation which involves a loop which refers to data residing off processor. The irregularly distributed arrays are `x` and `y`. Both the arrays have the same distribution pattern. Node 0 contains global indices 0, 1 and 2, while node 1 contains 3, 4, 5, 6 and 7. During the actual computation both nodes 0 and 1 needs to access certain elements of the `y` array. The global indices that node 0 has to access is 3, 7 and 1, and node 1 has to access 4, 2, 3, 0 and 6. Now we will present the inspector-executor code for the senario described above.

```
#define BLOCKED 1

int i,ndata,indirection;
int local[5],global_ref[5],local_ref[5];
double x[5],y[10];
TTABLE *tabptr;
SCHED *schedptr;


/* the following is the inspector code */

if(mynode() == 0){
  local[0] = 0;
  local[1] = 1;
  local[2] = 2;
  ndata = 3;
  tabptr = build_translation_table(BLOCKED,local,ndata);
  global_ref[0] = 3;
  global_ref[1] = 7;
  global_ref[2] = 1;
  localize(tabptr,&schedptr,global_ref,
                  local_ref,ndata,&n_off_proc,3);
} else {
  local[0] = 3;
  local[1] = 4;
  local[2] = 5;
  local[3] = 6;
  local[4] = 7;
  ndata = 5;
  tabptr = build_translation_table(BLOCKED,local,ndata);
  global_ref[0] = 4;
  global_ref[1] = 2;
  global_ref[2] = 3;
  global_ref[3] = 0;
  global_ref[4] = 6;
  localize(tabptr,&schedptr,global_ref,
```

```
                local_ref,ndata,&n_off_proc,5);
}


/* end of the inspector. Let us assign values to
   the distributed arrays                          */

for(i=0;i<ndata;i++){
   x[i] = i;
   y[i] = 2*i;
}

/* the following is the executor code */

dgather(schedptr,&y[ndata],y);

for(i=0;i<ndata;i++){
   indirection = local_ref[i];
   x[i] = x[i] + 3 * y[indirection];
}

/* end of the executor code */
```

After the end of the computation in processor 0 the values of x[0], x[1] and x[2] are 0.0, 25.0 and 8.0 respectively. On processor 1 the values of x[0], x[2], x[3], x[4] and x[5] are 6.0, 13.0, 2.0, 3.0 and 22.0 respectively. For a detailed example in FORTRAN refer to appendix B.

4 Calling the primitives from FORTRAN

This section shows how the primitives can be used with FORTRAN. We will go through the examples described in section 3 using the FORTRAN version of the PARTI primitives.

4.1 function ifschedule()

This function returns an integer which can be used to refer to the schedule corresponding to the input data.. This integer is used in gather, scatter and scatter_FUNC operations (Sections 4.2, 4.3, 4.4).

Synopsis

function ifschedule(ilocal,iproc,ndata)

Parameter declarations

integer ilocal() local indices to be gathered from or scattered to

integer iproc() processors to be gathered from or scattered to

integer ndata number of data elements involved in gather or scatter

Return value

Returns a reference to a schedule which can be used in PREFIXfgather, PREFIXfscatter, PREFIXfscatter_add, PREFIXfscatter_sub, PREFIXfscatter_mult.

Example

Node 0 schedules a fetch of elements 1 and 2 from a (so far unspecified) array on node 1; node 1 schedules a fetch of element 1 from an array on node 0 and 3 from an array on node 1.

```
logical ifschedule
integer ilocal(2), iproc(2), ndata
integer ischedinfo



if(mynode().eq.0){
  iproc(1)  = 1
  ilocal(1) = 1
```

```
  iproc(2)  = 1
  ilocal(2) = 2
  ndata = 2
}

if(mynode().eq.1){
  iproc(1)  = 0
  ilocal(1) = 1
  iproc(2)  = 1
  ilocal(2) = 3
  ndata = 2
}


ischedinfo = ifschedule(ilocal,iproc,ndata)
```

4.2 subroutine PREFIXfgather()

PREFIX can be d (double precision), i (integer) , f (real) or c (character). For more information refer to Section 3.4.

Synopsis

subroutine PREFIXfgather(ischedinfo,buffer,aloc)

Parameter Declarations

integer ischedinfo refers to the relevant schedule

TYPE buffer() pointer to buffer for copies of gathered data values

TYPE aloc() location from which data is to be fetched from calling processor

Return Value

None

Example

We assume that schedule has already been called with the parameters presented in Section 4.1. Our example will assume that we wish to gather double precision numbers, i.e. that we will be calling dfgather. On each processor, aloc points to the arrays from which values are to be obtained. buffer points to the location into which will be placed, copies of data values obtained from other processors.

```
      double precision buffer(2), aloc(3)
      integer ischedinfo


         do 10 i=1,3
           aloc(i) = mynode() + 0.1*i
10       continue

       call dfgather(ischedinfo,buffer,aloc)
```

On processor 0, buffer(1) and buffer(2) are now equal to 1.1 and 1.2. On processor 1, buffer(1) and buffer(2) are now equal to 0.1 and 1.3.

4.3 subroutine PREFIXfscatter()

PREFIX can be d (double precision), i (integer) , f (real) or c (character). For more information refer to Section 3.5.

Synopsis

subroutine PREFIXfscatter(ischedinfo,buffer,aloc)

Parameter Declarations

integer ischedinfo refers to the relevant schedule.

TYPE buffer() points to data values to be scattered from a given processor

TYPE aloc() points to first memory location on calling processor for scattered data

Return Value

None

Example

We assume that schedule has already been called with the parameters presented in Section 4.1. Our example will assume that we wish to scatter double precision numbers, i.e. that we will be calling dfscatter. On each processor, aloc points to the arrays to which values are to scattered. buffer points to the location from which will be obtained data that will be scattered The processor and local_array index to which the values are to be scattered was designated during an earlier call to schedule.

```
      double precision buffer(2), aloc(3)
      integer ischedinfo


        do 10 i=1,3
          aloc(i) = 10.0
10      continue

       if(mynode().eq.0) then
         buffer(1) = 444.44
         buffer(2) = 555.55
       endif

       if(mynode().eq.1) then
         buffer(1) = 666.66
```

```
  buffer(2) = 777.77
endif

call dfscatter(ischedinfo,buffer,aloc)
```

On processor 0, the first three elements of aloc are 666.66, 10.0 and 10.0. On processor 1, the first three elements of aloc are 444.44, 555.55 and 777.77.

4.4 subroutine PREFIXfscatter_FUNC()

PREFIX can be d (double precision), i (integer) , f (real) or c (character). For more information refer Section 3.6.

Synopsis

subroutine PREFIXfscatter_FUNC(ischedinfo,buffer,aloc)

Parameter Declarations

integer ischedinfo refers to the relevant schedule.

TYPE buffer() points to data values that will form operands for the specified type of remote operation.

TYPE aloc() points to first memory location on calling processor to be used as targets of remote operations.

Return Value

None

Example

We assume that schedule has already been called with the parameters presented in Section 4.1. Our example will assume that we wish to scatter and add double precision numbers, i.e. that we will be calling dfscatter_add. On each processor, aloc points to the arrays to which values are to be scattered and added. buffer points to the location from which will be obtained the values to be scattered and added. The processor and local_array index to which the values are to be scattered and added was designated during an earlier call to schedule.

```
      double precision buffer(2), aloc(3)
      integer ischedinfo


        do 10 i=1,3
          aloc(i) = 10.0
10      continue

       if(mynode().eq.0) then
         buffer(1) = 444.44
         buffer(2) = 555.55
       endif

       if(mynode().eq.1) then
         buffer(1) = 666.66
         buffer(2) = 777.77
       endif

       call dfscatter_add(ischedinfo,buffer,aloc)
```

On processor 0, the first three elements of aloc are 676.66, 10.0 and 10.0. On processor 1, the first three elements of aloc are 454.44, 565.55 and 787.77.

4.5 function ifbuild_translation_table()

For detailed information refer to Section 3.7.

Synopsis

function ifbuild_translation_table(part,indexarray,ndata)

Parameter Declarations

integer part how translation table will be mapped - may be BLOCKED or STRIPED

integer indexarray() each processor P specifies list of globally numbered indices for which P will be responsible

integer ndata number of indices for which processor P will be responsible

Return Value

integer which refers to the translation table corresponding to the input data.

Example

An example to demonstrate the use of both build_translation_table and dereference can be found in Section 4.7.

4.6 subroutine flocalize()

For more information refer to Section 3.9

Synopsis

subroutine flocalize(itabptr,ilsched,iglobal_refs, ilocal_refs,ndata,n_off_proc,my_size)

Parameter Declarations

integer itabptr refers to the relevant translation table pointer.

integer ilsched refers to the relevant schedule pointer (returned by `localize`).

integer iglobal_refs() the array which stores all the global reference string.

integer ilocal_refs() the array which stores the local reference string corresponding to the global references (returned by `localize`).

integer ndata number of global references.

integer n_off_proc number of off-processor data (returned by `localize`).

integer my_size the size of my local array.

Return Value

None

Example

Nodes 0 and 1 takes part in a computation which involves a loop which refers to data residing off processor. The inspector and the executor code is presented here.

```
      integer i,ndata,indirection
      integer local(5),iglobal_ref(5),ilocal_ref(5)
      double precision x(5),y(10)
      integer itabptr
      integer ischedptr
      logical ifbuild_translation_table


c the following is the inspector code

      BLOCKED = 1
      if(mynode().eq.0) then
        ilocal(1) = 1
        ilocal(2) = 2
        ilocal(3) = 3
        ndata     = 3
```

```
          mysize    = 3
          itabptr = ifbuild_translation_table(BLOCKED,ilocal,ndata)
          iglobal_ref(1) = 4
          iglobal_ref(2) = 8
          iglobal_ref(3) = 2
          call flocalize(itabptr,ischedptr,iglobal_ref,
                  ilocal_ref,ndata,n_off_proc,mysize)
        else
          ilocal(1) = 4
          ilocal(2) = 5
          ilocal(3) = 6
          ilocal(4) = 7
          ilocal(5) = 8
          ndata     = 5
          mysize    = 5
          itabptr = ifbuild_translation_table(BLOCKED,ilocal,ndata)
          iglobal_ref(1) = 5
          iglobal_ref(2) = 3
          iglobal_ref(3) = 4
          iglobal_ref(4) = 1
          iglobal_ref(5) = 7
          call flocalize(itabptr,ischedptr,iglobal_ref,
                  ilocal_ref,ndata,n_off_proc,mysize)
        endif
c
          do 10 i=1,ndata
            iglobal_ref(i) = ilocal_ref(i)
10        continue

c  end of the inspector. Let us assign values to
c   the distributed arrays

          do 20 i=1,ndata
            x(i) = i
            y(i) = 2*i
20        continue
```

```
c the following is the executor code

         call dfgather(ischedptr,y(ndata),y(1))

         do 30 i=1,ndata
           indirection = iglobal_ref(i)
           x(i) = x(i) + 3 * y(indirection)
30       continue

c end of the executor code
```

After the end of the computation in processor 0 the values of x(1), x(2) and x(3) are 25.0, 50.0 and 15.0 respectively. On processor 1 the values of x(1), x(2), x(3), x(4) and x(5) are 31.0, 20.0, 27.0, 10.0 and 47.0 respectively. For a detailed example in FORTRAN refer to appendix B.

4.7 subroutine fdereference()

For more information about this section refer to Section 3.8.

Synopsis

subroutine fdereference(index_table,global,local,proc,ndata)

Parameter declarations

integer index_table refers to the relevant translation table

integer global() list of global indices we wish to locate in distributed memory

integer local() local indices obtained from the distributed translation table that correspond to the global indices passed to dereference

integer proc() array of distributed translation table processor assignments for each global index passed to dereference

integer ndata number of elements to be dereferenced

Table 2: Values obtained by dereference

Processor	proc(1)	local(1)	proc(2)	local(2)
0	0	1	1	1
1	1	2	0	2

Return value

None

Example

A one dimensional distributed array is partitioned in some irregular manner so we need a distributed translation table to keep track of where one can find the value of a given element of the distributed array.

In the example below, we initialize a translation table. Processor 0 calls build_translation_tabl and assigns indices 1 and 4 to processor 0, processor 1 calls build_translation_table and assigns indices 2 and 3 to processor 1. The translation table is partitioned between processors in blocks.

Processor 0 then uses the translation table to dereference global variables 1 and 2, processor 1 uses the translation table to dereference global variables 3 and 4. On each processor, dereference carries out a translation table lookup. The values of proc and local are returned by dereference are shown in Table 2). The user gets to specify the processor to which each global index is assigned, note however that build_translation_table assigns local indices.

```
      program dref
c
      integer size, i, index_array(2)
      integer ideref_array(2)
      integer ilocal(2), iproc(2)
      logical ifbuild_translation_table



c     Assign indices 1 and 4 to processor 0
```

```
      if(mynode().eq.0) then
        index_array(1) = 1
        index_array(2) = 4
      endif
c   Assign indices 2 and 3 to processor 1

      if(mynode().eq.1) then
        index_array(1) = 2
        index_array(2) = 3
      endif

c   set up a translation table

      BLOCKED = 1
      size    = 2
      itable = ifbuild_translation_table(BLOCKED,index_array,size)

c    Processor 0 seeks processor and local indices
c     for global array indices 0 and 1 */

      if(mynode().eq.0) then
        ideref_array(1) = 1
        ideref_array(2) = 2
      endif

c    Processor 1 seeks processor and local indices
c     for global array indices 2 and 3 */

      if(mynode().eq.1) then
        ideref_array(1) = 3
        ideref_array(2) = 4
      endif

c   Dereference a set of global variables

      call fdereference(itable,deref_array,local,proc,size)
```

```
c local and proc return the processors and local indices where
c global array indices are stored.
c In processor 0, proc(1) = 0, proc(2) = 1, local(1) = 0 , local(2) = 0
c In processor 1, proc(1) = 1, proc(2) = 0, local(1) = 1 , local(2) = 1
      stop
      end
```

Now assume that processor 0 needs to know to values of distributed array elements 1,2, and 4 while processor 1 needs to know the value of element 3. We call dereference to find the processors and the local indices that correspond to each global index. At this point schedule can be called and gathers and scatters carried out.

5 Acknowledgements

We would like to thank Seema Hiranandani, Jeff Scroggs and Janet Wu for their help in debugging the primitives presented here. We also thank Janet Wu for her formulation of the build_translation_table primitive. We thank Dimitri Mavriplis for his continuing input during the development of the primitives and also for letting us use his code in the FORTRAN example shown in the appendix. We also thank Eugene Poole and Nahil Sobh for letting us use their matrix-vector multiplication code shown in the appendix. We would like to thank Adam Rifkin for his careful proofing of this manual. Finally, we would like to thank Bob Voigt and Martin Schultz for their support during this project's long (and continuing) incubation period. It takes time to put together a good PARTI!

A Sweep over the Edges of an Unstructured Mesh

This code can be found in the directory `examples/unst`. This goes through the whole process of setting up the inspector and then the subroutine executor is called to do the actual computation. There is a driver program which is included in the distribution but not added in this section. The executor is a loop which has been taken out of a real CFD code, where the loop is over the edges of the mesh. In the subroutine

executor, if we remove the calls to gather and scatter_add then the piece of code looks identical to the sequential version.

```
c-------------------------------------------------------------
c The subroutines inspector and executor for sweep over an
c arbitrary unstructured mesh is shown below.
c
c There is a driver code which calls these two subroutines after
c reading in the mesh structure and initialization data. This
c shows how the different PARTI primitives can be called
c from FORTRAN.
c
c-------------------------------------------------------------


c-----------------------------------------------------
       subroutine inspector(ledge,myvals,nde)
c-----------------------------------------------------
c
c---------------------------------------
#include "common1.F"
c---------------------------------------
       common/node/ ntotnodes,nonode,noedge
       common/sched/ lesched
       common/offproc/ ne_off_proc
c
       integer nde(ledge,2)
       integer myvals(nonode)
c
c------ Local Variables
c
       integer ig_ref_e(nge)
       integer locale(nge)
       logical ifbuild_translation_table
c
c------ Build the translation table
```

```
c
       itabptr = ifbuild_translation_table(1,myvals,nonode)
c
c------ Setup global references for edge loop
c
       do 20 i = 1,noedge
         ig_ref_e(i) = nde(i,1)
         ig_ref_e(noedge+i) = nde(i,2)
  20   continue
       iecount = 2 * noedge
c
c------ Setup schedule and change global ref. to local ref.
c
      call flocalize(itabptr,lesched,ig_ref_e,locale,
     .                              iecount,ne_off_proc,nonode)
c
       do 40 i = 1,noedge
         nde(i,1) = locale(i)
         nde(i,2) = locale(noedge+i)
  40   continue
c
       return
       end

c
c
c
c-----------------------------------------------------------------
c
      subroutine executor(ledge,lnode,nde,gnorm,w,p,dtl,iflop)
c
c-----------------------------------------------------------------
c
      real*8 rm,al,yaw,gamma,rho0,p0,ei0,h0,c0,u0,v0,w0
      real*8 cfl,bc,vis0,vis1,vis2,hm,smoop
c
      common/node/ ntotnodes,nonode,noedge
```

```
      common/sched/ lesched
      common/offproc/ ne_off_proc
      common/tsp/ cfl,bc,vis0,vis1,vis2,hm,smoop,ncycsm
      common/flw/ rm,al,yaw,gamma,rho0,p0,ei0,h0,c0,u0,v0,w0
c
      integer nde(ledge,2)
      real*8  gnorm(ledge,5)
      real*8  dtl(lnode)
      real*8  w(lnode,5),p(lnode)
c
c--Local variables
c
      real*8  cc1,cc2,cs1,cs2,a1,a2,qs,flux1,flux2

c
c--Initialize Time Step
c
      do 50 i=1,nonode
      dtl(i)   = 0.0D0
   50 continue
c
c-- Do all the Gathers
c
      do 60 kk = 1,4
        call dfgather(lesched,w(nonode+1,kk),w(1,kk))
   60 continue
      call dfgather(lesched,p(nonode+1),p(1))
      do 63 i = 1,ne_off_proc
        dtl(nonode+i) = 0.0D0
   63 continue
c
c--Compute Field Time-Steps Using Edge Format
c
      do 500 i=1,noedge
      n1       = nde(i,1)
      n2       = nde(i,2)
      cc1      = dsqrt(gamma*p(n1)/w(n1,1))
```

```
      cc2       = dsqrt(gamma*p(n2)/w(n2,1))
      cs1       = cc1*gnorm(i,4)
      cs2       = cc2*gnorm(i,5)
      a1        = (gnorm(i,1)*w(n1,2) + gnorm(i,2)*w(n1,3)
     .                               + gnorm(i,3)*w(n1,4)) / w(n1,1)
      a2        = (gnorm(i,1)*w(n2,2) + gnorm(i,2)*w(n2,3)
     .                               + gnorm(i,3)*w(n2,4)) / w(n2,1)
      qs        = (a1 + a2) / 2.0D0
      flux1     = dabs(qs) + cs1
      flux2     = dabs(qs) + cs2
      dtl(n1)   = dtl(n1) +   flux2
      dtl(n2)   = dtl(n2) +   flux1
  500 continue
      iflop = iflop + (noedge * 28)
c
c-- Do all the Scatters
c
      call dfscatter_add(lesched,dtl(nonode+1),dtl(1))
c
      return
      end
```

B Example : Sparse matrix multiplication

The following example of symmetric matrix vector multiplication can be found in the file `matmult.c` in the `examples/sparse_mat_mult` directory. There is a host program which is present in the same directory but has not been listed here. The sparse matrix is obtained from the host program using the function `get_sparse_mat()`. Then we go through the pre- processing to generate all the fetch lists and build a schedule to bring in off-processor data. Lastly, the matrix multiplication procedure `spmvm()` is called. After the multiplication the values are scattered using the primitive `scatter_add`

```
/*************************************************************/
/* PARTI program to do a sparse matrix-vector multiplication */
```

```
/*                                                              */
/* This program reads in a sparse matrix with the help of       */
/* the host program and does a matrix vector multiplication.    */
/* The is a listing of the node program and it is run by the    */
/* host program. This program:                                  */
/*                                                              */
/*     1) gets unstructured mesh (w/ help from host program)    */
/*     2) does lots of memory and address stuff on it           */
/*     3) generates a vector x                                  */
/*     4) multiplies x by the matrix, getting y                 */
/*                                                              */
/****************************************************************/


#include <cube.h>
#include <stdio.h>
#include <math.h>
#include "parti.h"
#include "main.h"
main(argc,argv)
int argc;
char *argv[];
{
  int i, j, count;
  TTABLE *table;
  SCHED *sr;
  double *x, *y, *z;

  /*
   * ------------------------------------
   * Get sparse matrix from host program.
   * ------------------------------------
   */

  get_sparse_mat();

  /*
```

```
     * ------------------------------------------------------
     * Build translation table by scattering Row to the table.
     * IN: Row[i]       OUT: table
     * ------------------------------------------------------
     */

    table = build_translation_table(BLOCKED,Row,Myrows);

    /*
     * ------------------------------------------------------
     * Look up address of Cols and put them in Local and Proc.
     * IN: Cols[i],table    OUT: Local[i],Proc[i]
     * ------------------------------------------------------
     */

    dereference(table,Cols,Local,Proc,Mynonzeros);

  /*
   * ------------------------------------------------------
   * Loop through all proc/offset pairs and decide which
   * must be fetched from other processors.
   * IN: Local[i],Proc[i]      OUT: Fetch_l[i],Fetch_p[i]
   * ------------------------------------------------------
   */

    gen_fetch_list();

  /*
   * ------------------------------------------------------------
   * Allocate memory for vectors , and set x[i] = i for local i.
   * ------------------------------------------------------------
   */

    x = (double *) malloc(sizeof(double)*Myrows);
    y = (double *) malloc(sizeof(double)*Myrows);

    for(i=0;i<Myrows;i++) x[i] = 1.0;
```

```
/*
 * ----------------------------------
 * build communications schedule
 * IN: Fetch_l[i],Fetch_p[i]   OUT: sr
 * ----------------------------------
 */

  sr = schedule(Fetch_l,Fetch_p,Nfetch);

/*
 * ------------------------------------------
 * Perform sparse-matrix vector multiplication.
 * ------------------------------------------
 */

  spmvm(sr,x,y);
}
 /* END OF NODE PROGRAM */

/*
 * ------------------------------------------------
 * This function is used to read in the sprse mat.
 * It should be ignored if at all possible.
 * ------------------------------------------------
 */

get_sparse_mat()
{
  int size, indx_buffer[BUFFER_SIZE];
  double coef_buffer[BUFFER_SIZE];
  int type, rows_expected;

  rows_expected = -1;
  Myrows = 0;
  Mynonzeros = 0;
```

```
  gsync();
  while( (Myrows<rows_expected) | (rows_expected<0) ){
         cprobe(-1);
         type = infotype();
         size = infocount()/sizeof(int);
         if( type==ROW_INDX_MSG ){
            crecv( ROW_INDX_MSG,indx_buffer,size*sizeof(int));
            crecv( ROW_COEF_MSG,coef_buffer,size*sizeof(double));
            unpack_row_data(indx_buffer,coef_buffer,size);
         }
         if( type==SETUP_MSG ){
            crecv(SETUP_MSG,indx_buffer,size*sizeof(int));
            rows_expected = indx_buffer[mynode()];
            Nrows = indx_buffer[numnodes()];
         }
  }
  gsync();

}

/*
 * ----------------------------------------
 * The buffers are unpacked in the following
 * procedure
 * ----------------------------------------
 */

unpack_row_data(indx_buffer,coef_buffer,size)
int *indx_buffer,size;
double *coef_buffer;
{
  int count, i, j, row, ncols, count2, ixx, ist;
  double sum;
  static int col_count = 0;

  for( count=0; count<size; ){
       Row[Myrows] = indx_buffer[count];
```

```
        Diags[Myrows] = coef_buffer[count];
        sum=Diags[Myrows];
        ncols = Ncols[Myrows] = indx_buffer[count+1];
        count=count+2;
        Mynonzeros += ncols;

        if( Myrows >= MAX_ROWS ){
            fprintf(stderr,"Error on node %d : too many rows!!!\n",mynode());
            exit();
        }

        if( Mynonzeros >= MAX_NONZEROS ){
            fprintf(stderr,"Error on node %d : too many nonzeros!!!\n",
               mynode());
            exit();
        }

        for( j=0; j<ncols; j++){
            Cols[col_count] = indx_buffer[count];
            Vals[col_count] = coef_buffer[count];
            sum+=Vals[col_count];
            col_count++;
            count++;
        }
        Myrows++;
    }
  }

/*
 * ------------------------------------------------
 * This function takes the Locol[i],Proc[i]
 * address for each nonzero col in the matrix
 * and puts nonlocal ones into Fetch_l[i],Fetch_p[i]
 * ------------------------------------------------
 */

gen_fetch_list()
```

```
{
  int count, i, myproc;

  myproc = mynode();
   /* count offnode refs.  */
  Nfetch = 0;
  for( i=0; i<Mynonzeros; i++) Nfetch += (Proc[i]!=myproc);
   /* for each ref.   */
  Fetch_p = (int *) malloc(sizeof(int)*Nfetch*2);
  Fetch_l = &Fetch_p[Nfetch];
  count = 0;
  for( i=0; i<Mynonzeros; i++ ){
       if( Proc[i] != myproc ){
         /* if Col[i] refers to an off-proc location.. */
           Fetch_p[count] = Proc[i]; /* add it to the fetch list */
           Fetch_l[count] = Local[i];
          count++;
         }
      }
}

/*
 * ------------------------------------------------
 * sparse matrix vector multiply function !
 * require that the schedule be built and passed in
 * ------------------------------------------------
 */

spmvm(sr,x,y)
SCHED *sr;     /*  communication schedule */
double *x, *y; /*  input and result vectors */
{
  int myproc, bcount, count, i, j;
  double tmp, *buffer, *ybuffer;

   /* Allocate local buffer to gather data into. */
  buffer = (double *) malloc(sizeof(double)*Nfetch);
```

```
 /* Allocate local buffer to store output vector values into. */
ybuffer = (double *) malloc(sizeof(double)*Nfetch);
 /* Gather data using previously computed communication schedule. */
dgather(sr,buffer,x);

myproc = mynode();
bcount = 0;
count = 0;
for( i=0; i<Myrows; i++ ) y[i]=0.0;
for( i=0; i<Nfetch; i++ ) ybuffer[i]=0.0;

for( i=0; i<Myrows; i++ ){
     y[i] += Diags[i]*x[i];
     for( j=0; j<Ncols[i]; j++ ){
        /* for each nonzero col .... */
        if( Proc[count] == myproc ){
             /* if col[count] is local */
              y[i] += x[Local[count]]*Vals[count];
              y[Local[count]] += x[i]*Vals[count];
        } else {
             /* otherwise look in buffer */
              y[i] += buffer[bcount]*Vals[count];
              ybuffer[bcount] += x[i]*Vals[count];
              bcount++;
        }
          count++;
     }
}
dscatter_add(sr,ybuffer,y);
gsync();

for( i=0; i<Myrows; i++ ){
     fprintf(myfile," after scatter processor %d, y[%d] = %lf\n",
            myproc,i,y[i]);
     fflush(myfile);
}
free(buffer);
```

```
  free(ybuffer);
}
```

References

[1] H. BERRYMAN, J. SALTZ, AND J. SCROGGS, *Execution time support for adaptive scientific algorithms on distributed memory machines*, ICASE Report 90-41, May 1990.

[2] R. MIRCHANDANEY, J. H. SALTZ, R. M. SMITH, D. M. NICOL, AND K. CROWLEY, *Principles of runtime support for parallel processors*, in Proceedings of the 1988 ACM International Conference on Supercomputing , St. Malo France, July 1988, pp. 140–152.

[3] S. MIRCHANDANEY, J. SALTZ, P. MEHROTRA, AND H. BERRYMAN, *A scheme for supporting automatic data migration on multicomputers*, in Proceedings of the Fifth Distributed Memory Computing Conference, Charleston S.C., 1990.

[4] J. SALTZ, H. BERRYMAN, AND J. WU, *Runtime compilation for multiprocessors*, ICASE Report 90-59, 1990.

[5] J. SALTZ, K. CROWLEY, R. MIRCHANDANEY, AND H. BERRYMAN, *Run-time scheduling and execution of loops on message passing machines*, Journal of Parallel and Distributed Computing, 8 (1990), pp. 303–312.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-187568 ICASE Interim Report 17		

4. Title and Subtitle	5. Report Date
A MANUAL FOR PARTI RUNTIME PRIMITIVES Revision 1	May 1991
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Raja Das Joel Saltz Harry Berryman	Interim Report No. 17
	10. Work Unit No.
	505-90-52-01

9. Performing Organization Name and Address	11. Contract or Grant No.
Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225	NAS1-18605
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Contractor Report
National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
Michael F. Card

Final Report

16. Abstract

Primitives are presented that are designed to help users efficiently program irregular problems (e.g. unstructured mesh sweeps, sparse matrix codes, adaptive mesh partial differential equations solvers) on distributed memory machines. These primitives are also designed for use in compilers for distributed memory multiprocessors. Communications patterns are captured at runtime, and the appropriate send and receive messages are automatically generated.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
sparse, unstructured, library, PARTI	61 - Computer Programming and Software 64 - Numerical Analysis Unclassified - Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	52	A04